RAPPORT TP1 DIMITRI NICOLAS, mEHDI ABERKANE

# Exercice 1

**Fonctions d’inversion (reverse) :**

(defun reverseA(arg1 arg2 arg3)

    (list arg3 arg2 arg1)

)

La fonction n’admettant que 3 arguments, nous pouvons directement retourner une liste inversée de ces éléments. Cela est tout à fait gérable pour trois éléments.

(defun reverseB(L)

    (let ((listReversed '()))

    (if (listp L)

        (dolist (elem L)

            (push elem listReversed)

        )

        (print "Erreur : L n'est pas une liste")

    )

    listReversed

))

Pour l’inversion d’une liste à taille variable, nous avons développé deux versions (ici itérative) dont celle présentée repose sur une liste locale que l’on alimentera par ajout des éléments de la liste passée en paramètre au début.

(defun reverseC(L)

    (if (listp L)

        (if (> (length L) 0)

            (append (reverseC (cdr L)) (list (car L)))

        )

        (print "Erreur : L n'est pas une liste")

    )

)

(defun monReverse(L)

    (if (listp L)

        (if (> (length L) 0)

            (append (reverseC (cdr L)) (list (car L)))

        )

        (print "Erreur : L n'est pas une liste")

    )

)

Cette version, plus concise repose sur la récursivité en allant récupérer le premier élément (« car ») des listes ôtées récursivement de leur premier élément (cdr). Ce « car » que l’on transforme en liste (pour assurer une bonne concaténation), après l’appel récursif (pour remonter la liste en sens inverse).

**Fonctions de traitement sur les listes :**

(defun double(L)

    (if (listp L)

        (if (> (length L) 0)

            (append (list (car L)) (if (atom (car L)) (list (car L))) (double (cdr L)))

        )

        (print "Erreur : L n'est pas une liste")

    )

)

Pour cette fonction de doublage, nous nous reprenons le principe de la version récursive de reverseC en ajoutant le « car » au début puis en ajoutant une seconde fois ce « car » si jamais il s’agit d’un atome.

(defun nombres3 (L)

    (if (listp L)

        (if (and (numberp (car L)) (numberp (cadr L)) (numberp (caddr L)))

            "BRAVO"

            "PERDU"

        )

        (print "Erreur : L n'est pas une liste")

    )

)

De la même manière que pour la première fonction d’inversement, ne s’agissant que de trois éléments à vérifier, on peut se contenter d’évaluer le « car », « cadr » et « caddr ».

(defun grouper(L1 L2)

    (let ((groupedList '()))

    (if (= (length L1) (length L2))

        (loop for idx from 1 to (length L1)

            do (push (list (nth (- (length L1) idx) L1) (nth (- (length L2) idx) L2)) groupedList)

        )

        (print "Erreur: Les deux listes sont de taille différente")

    )

    groupedList

))

Afin de grouper les éléments des deux listes, nous allons ici parcourir l’une d’entre elle (on choisit arbitrairement L1) à l’aide de son indice. On va ensuite prendre les éléments d’indice (taille – index courant) de L1 et L2, les regrouper dans une liste et les insérer en tête de la liste finale (d’où l’utilité de prendre l’élément d’indice taille – index parcouru).

(defun palindrome(L)

    (if (listp L)

        (equal (reverse L) L)

        (print "Erreur : L n'est pas une liste")

    )

)

Vérifier si la liste passée est un palindrome revient à vérifier si la liste de départ est équivalente à son inverse. Ici, nous utilisons la fonction native au LISP « reverse », nous aurions également pu utiliser nos fonctions développées précédemment.

# Exercice 2

(defun list-triple-couple(L)

    (if (listp L)

        (mapcar (lambda (x) (list x (\* 3 x))) L)

        (print "Erreur : L n'est pas une liste")

    )

)

Pour chaque élément de la liste L, nous effectuerons l’expression renvoyant une liste composée de x et de son triple.

# Exercice 3

(defun cles(a-list)

    (if (listp a-list)

        (mapcar (lambda (assoc)

            (car assoc)

        ) a-list)

    )

)

Les clés des associations représentant le premier élément de l’association (correspondant à une liste), pour chaque association nous allons prendre le premier élément à savoir sa clé ce qui retournera finalement la liste des clés.

(defun creation(listeCles listeVal)

    (grouper listeCles listeVal)

)

Pour pouvoir créer une liste d’associations, on va se servir de notre fonction « grouper » pour réunir les clés et les valeurs entre-elles sous forme d’associations distinctes.

(defun my-assoc (cle a-list)

    (if (listp a-list)

        (mapcar (lambda (x)

            (if (not (equal (member cle x) nil))

                (member cle x)

            )

        ) a-list)

        (print "Erreur : a-list n'est pas une liste")

    )

)

Dans chaque association de a-list, nous allons chercher si la clé passée en paramètre est présente, auquel cas on retourne l’association correspondante.

Aussi, nous avons vérifié quand c’était nécessaire les différents tests sur les paramètres (taille définie, paramètre est bien une liste).

# Exercice 4

Rappel sur la structure d’une liste recensant les informations d’une tombe :

Une liste tombe contient cinq éléments. Le premier correspond au nom et est de format string. Le second est un entier représentant l’année d’inhumation. Le troisième est une liste comportant deux éléments : respectivement la rangée et le numéro de la tombe. Le quatrième est l’année de début de location et le dernier élément est un entier qui correspond à la durée de location.

Exemple : ( « Bécaud » 2001 (45 17) 2000 30)

Préliminaire :

Afin d’assurer que les fonctions de la question 1 ne provoqueront pas d’erreur, nous avons choisi d’écrire un prédicat simple  ***bon-format*** qui vérifie que la tombe passée en paramètre est bel et bien une liste comportant 5 éléments dont le troisième est une liste. Ce prédicat ***bon-format***  est utilisé dans toutes les fonctions de la question 1, elles-mêmes appelées par la suite. La vérification est volontairement simple car on suppose que l’utilisateur cherche à entrer des bons formats.

**Question 1:**

Pour chacune de ces fonctions qui consistent à récupérer uniquement le nième élément de la tombe, nous avons procédé comme suit :

La fonction vérifie si la tombe passée est au bon format.

**Si elle l’est:**

Alors elle renvoie l’élément de la liste qui correspond à l’élément demandé.

Pour renvoyer l’élément demandé, nous avons employé deux méthodes toutes deux a priori également efficaces.

Première méthode ( dans : nom, num, rangée) : composer les fonctions car et cdr pour obtenir l’élément souhaité. Par exemple, pour récupérer le nom dans la liste de la tombe, on retourne le ***car*** de la liste tombe ( le premier élément). Pour récupérer l’année d’inhumation, on récupère donc le premier élément de la queue de la liste, soit le car du cdr, on appelle donc la fonction ***cadr*** en donnant tombe comme paramètre.

Seconde méthode : lorsque l’on veut obtenir le nième élément d’une liste L, on utilise la méthode

(***nth*** n-1 L) car les indices des éléments d’une liste commencent à 0.

**Si elle ne l’est pas :**

On retourne « Erreur format tombe. ».

**Question 2:**

On cherche ici à trouver le nom d’un défunt enterré dans un cimetière à partir de son emplacement. On écrit donc la fonction qui-est-la qui prend en paramètre le cimetière et l’emplacement sous forme d’une liste de deux entiers.

(Qui-est-la cimetière emplacement)

Dans l’exemple, le cimetière prenait la forme d’une liste de deux éléments, le premier étant le nom du cimetière et le second étant un élément contenant la liste des tombes.

Nous avons écrit une fonction récursive. Son fonctionnement est le suivant :

**Si le format du premier élément est une liste de tombe**, alors la fonction compare l’emplacement du premier élément de la liste avec l’emplacement recherché. En cas d’égalité, la fonction retourne le nom du défunt. En cas d’inégalité, la fonction se rappelle avec pour paramètre le cdr du cimetière et l’emplacement pour tester l’élément suivant.

**Si le format du premier élément n’est pas une liste de tombe,** alors la fonction vérifie si le second élément de la liste est une liste s’il existe. Si cette condition est vérifiée, alors on rappelle la fonction avec le cdr du cimetière et le même emplacement. Sinon, on affiche emplacement non attribué car cela signifie que l’on fait un appel avec une liste vide ou qui ne correspond pas à un cimetière.

Remarque : Lorsque l’on cherche le nom et l’emplacement d’une tombe, on a recours aux fonctions rédigées précédemment.

**Question 3 :**

On cherche à vérifier si la tombe a été réservée depuis une année antérieure à l’inhumation du défunt, autrement dit s’il ( ou quelqu’un d’autre) s’est montré prévoyant. On écrit donc le prédicat prévoyant? Qui prend en paramètre une tombe.

**Pour éviter l’utilisation d’une structure de contrôle ( if )**, on va utiliser la fonction primitive > de lisp, qui s’utilise comme suit: (> entierA entierB). Si entierA>entierB la fonction retourne t, sinon nil.

Alors la seule ligne que nous avons écrit à l’intérieur de la fonction est celle-ci :

(> (an-inhum tombe) (debut-loc tombe))

Le prédicat retournera nil ou t car il retournera ce que le prédicat > retournera.

**Question 4 :**

Cette fois-ci, on veut compter pour un cimetière le nombre de défunts qui ont été « prévoyants ».

On écrira la fonction (nb-prevoyants cimetière) qui prend en paramètre un cimetière et qui retourne le nombre de personnes ayant une année d’inhumation strictement ultérieure à leur date de début de location.

Pour se faire, on va initialiser à 0 une variable locale avec let qui agira en compteur. On initialise une seconde variable locale qui sera égale au cadr du cimetière, soit à la liste des tombes. ( on rappelle qu’un cimetière est modélisé de cette forme ( nom\_cimetiere ((tombe1) … (tombei))).

On parcourt l’ensemble de la liste des tombes avec la boucle loop qui fait varier i de 0 à la taille de la liste-1 ( rappel : l’index commence à 0). Ensuite, on vérifie pour la tombe stockée au rang i si le défunt a été prévoyant en utilisant notre prédicat (cf Q.3). Si le défunt est prévoyant, alors on incrémente le compteur de 1.

Après la boucle, on retourne le compteur.

**Question 5 :**

On veut cette fois-ci retrouver dans un cimetière la liste de tous les noms des défunts enterrés à une certaine rangée.

On écrira la fonction annuaire qui prend en paramètre un cimetière et une rangée, qui retourne une liste de noms.

Les tombes ne sont pas triés par rangées, il faudra donc parcourir l’ensemble de la liste.

Par conséquent, la logique de cette fonction va ressembler à la précédente.

On parcourt la liste des tombes également avec la boucle loop. Cette fois-ci, on initialise une variable la liste des noms cherchés.

Lors du parcours de la liste, on vérifie pour chaque tombe la valeur de sa rangée en utilisant la fonction rangée (cf Q.1). Si la tombe est présente dans la rangée recherchée, alors on va concaténer à la liste des noms le nom de la tombe, récupéré via la fonction nom codée en 1.

Question 6 :

L’objectif de cette question est de chercher une solution élégante et efficace pour retrouver le doyen et le benjamin d’un cimetière.

Nous avons choisi d’initialiser deux variables locales doyen et benjamin toutes deux initialisées au car du cimetière passé en paramètre.

On parcourra la liste avec la boucle loop. Pour chaque tombe, si son inhumation a eu lieu avant celle stockée dans la variable doyen, alors cette dernière prend la valeur de la tombe courante. De même, si son inhumation a eu lieu après celle du benjamin courant, benjamin prend la valeur de la tombe courante.

Après la boucle, la fonction retourne la liste composée du doyen et du benjamin. (List doyen benjamin).

NOTE :

Dans les exemples, les fonctions étaient parfois appelés avec des paramètres supplémentaires que nous n’avons pas compris. (‘Cimetières) .

Pour obtenir un résultat qui nous semblait le plus pertinent, nous avons choisi de prendre en paramètre des cimetières tels que Père Lachaise et répondre à la consigne. Nous testions nos fonctions sans la quote avec la variable PERELACHAISE que nous avons initialisé avec les données en exemple proposées dans le TP.

Chacune des fonctions retourne le résultat que nous escomptions et nous nous sommes attachés à produire les solutions les plus simples, élégantes et efficaces.